CuPIDS: An Exploration of Highly Focused, Co-processor-based Information System Protection

Paul D. Williams, Eugene H. Spafford

CERIAS, Purdue University, West Lafayette, IN

Abstract

The Co-Processing Intrusion Detection System (CuPIDS) project is exploring how tom improve information system security by dedicating computational resources to system security tasks in a shared resource, multi-processor (MP) architecture. Our research explores ways in which this architecture offers improvements over the traditional uni-processor (UP) model of security. There are a number of areas to explore, one of which has a protected application running on one processor in a symmetric multiprocessing (SMP) system while a shadow process specific to that application runs on a different processor, monitoring its activity, ready to respond immediately if the application violates policy. Experiments with a prototype CuPIDS system demonstrate the feasibility of this approach. Fine-grained protection of the real-world application WU-FTP resulted in less than a ten percent slowdown while demonstrating CuPIDS' ability to quickly detect illegitimate behavior, raise an alarm, automatically repair the damage done by the fault or attack, allow the application to resume execution, and export a signature for the activity leading up to the error.

Key words: Intrusion detection, Information System Security, Co-processor, Multi-processor
PACS:

1 Introduction

This paper describes research into the Co-Processing Intrusion Detection System (CuPIDS)–an exploration into increasing information system security by dedicating computational resources to system security tasks in a shared resource, multi-processor (MP) architecture. We believe this architecture will allow IDS to use higher fidelity monitoring models, particularly with regard to the timeliness of detection, and will also increase system robustness in the

face of some types of attacks. Our philosophical foundations are fourfold: high assurance is important, a great deal of information about how systems are supposed to operate is often available but rarely used, MP computer systems are becoming commonplace, and finally that information systems will be vulnerable to attack or erroneous behavior for the foreseeable future.

While a body of research into co-processing techniques for tasks such as secure booting and digital rights management exists [1,2], not nearly as much work has been done in investigating how generalized security tasks can benefit from dedicated co-processing. Most past and present Intrusion Detection System (IDS) architectures assume a uni-processor environment, or do not explicitly make use of multiple processors when they exist. The advent of multicore processors from the mainstream processor manufacturers such as Sun, Intel and AMD will result in MP systems becoming more common outside the server farm. We believe this affords us novel opportunities to be creative with how system resources are allocated.

We are concerned with a very general threat model that assumes:

- Processes running at any privilege level in the production parts of the system may be compromised at any time after boot is complete.
- Attacks or faults may be caused by the activities of local or external users or a combination of both.
- Attacks or faults may result in a system compromise without ever causing a context switching event.

We believe that under some circumstances CuPIDS can be more effective than Standard Uni-processor-based Intrusion Detection/Intrusion Prevention Systems (StUPIDS) [1].

For our purposes more effective is shown by demonstrating that:

(1) Running concurrently with attack code affords CuPIDS opportunities to detect and respond to attacks that are not available to StUPIDS.
(2) Because the opportunity exists to detect attacks while they occur without waiting for a context-switching event (either between user processes or between user and kernel mode) CuPIDS may be able to respond more quickly and attacks may be detected with higher fidelity.

These are advantages that are difficult or impossible to achieve on a uni-processor system—no matter how powerful.

[1] The name StUPIDS is in tribute to the work done in Purdue's Coast Laboratory on the IDIOT intrusion detection system[3].

2 Background

This section describes the time and intrusion detection domain with which we are concerned and briefly references related research.

2.1 Time Domain

Because some of the primary gains we anticipate from CuPIDS are time-related, we need to clarify what time domain we are working in. To do so we draw from a recent categorization of computer security systems. Kuperman's Ph.D. dissertation [4] describes four major timeliness categories in which detection can be accomplished: real-time, near real-time, periodic and retrospective. It is in the category of real-time and near real-time that CuPIDS offers significant gains over StUPIDS.

To specify what we mean by real-time and near real-time we borrow Kuperman's notation. We represent the set of events taking place in a computer system by the set E. This set contains suspect events B such that $B \subseteq E$ and there exist events a, b, and c such that $a, b, c \in E$ and $b \in B$ The notation t_x represents the time of occurrence of event x. Finally, we need a detection function $D(x)$ that determines the truth of the statement $x \in B$.

Real-time: Detection of a bad event b takes place while the system is operating and is further restricted to mean that detection of b occurs before an event, c, dependant upon b takes place. Given E, real-time detection requires the ordering

$$t_a < t_{D(b)} < t_c$$

Near real-time: Detection of a bad event b occurs within some, typically small, finite time δ of the occurrence of b. This requires the ordering

$$|t_b - t_{D(b)}| \leq \delta$$

While no complete detection function $D(x)$ exists, there are a great number of bad events, $B_D = \{b_0, b_1, \ldots, b_n\} \in B$ for which we do have effective detection functions. Assuming the existence of identical CuPIDS and StUPIDS detection functions, $D_{CuPIDS}(B_D)$ and $D_{StUPIDS}(B_D)$ CuPIDS offers improvements in guaranteed detection time. On a uni-processor system in which the StUPIDS runs as a normal task the soonest it can possibly detect a bad event, b_i, is when a context switching event occurs after t_{b_i} but before t_{c_i} and the scheduler chooses the StUPIDS to run. In the best case b_i involves the execution of a system call or some other blocking event, the scheduler picks the appropriate StUPIDS process to run next, and b_i is detected before c_i can occur. In the

worst case the system is compromised before the StUPIDS has an opportunity to run and detect b_i.

Other complications include the relative priority of StUPIDS processes to other processes in the system, and even if a StUPIDS process is chosen to run, its portion of $D_{StUPIDS}(B_D)$ may not include b_i. Therefore even though the StUPIDS is capable of detecting b_i it may not do so before the production process is made active again and t_{c_i} occurs. This means that even though $D_{StUPIDS}(b_i)$ exists a StUPIDS can at best claim near-realtime detection with $\delta = CPUQuantum$. In the case of a StUPIDS running on a MP machine, the appropriate monitoring process may be executing at the right time; however, there is no guarantee that this is the case. CuPIDS reduces the uncertainties described above by ensuring, whenever possible, the appropriate monitor is executing, thus offering real-time detection capability.

2.2 Detection Domain

Among the factors that make intrusion detection in generalized computing environments difficult is the wide range of capabilities that must be protected. By forcing the security system designer to cover a wider range of resources, the defensive assets are, in a sense, "stretched thinner" than they will be in the highly focused CuPIDS environment. CuPIDS' ability to concentrate the right defenses at the right time on critical tasks coupled with the ability to use well defined security boundaries as defined by the program designer and system security policy allows the exploration of highly effective intrusion detection functions.

While our research is generally applicable to any computing environment in which multiple processors are available, we anticipate that it will be most useful in the dedicated server environment. Ideally, these machines are not used for general purpose computing and run only a streamlined set of applications dedicated to the service the system provides. These simplified configurations are not only simpler to maintain, but their smaller attack surfaces [5] are simpler to defend as well.

2.3 Prior Research

There exists an enormous body of work into techniques for detecting and preventing violations of security policy. Axelsson's in-depth, thorough taxonomy and survey of the field of intrusion detection in 2000 is a good starting point for those unfamiliar with the field [6]. We draw from those techniques and augment them in ways that make use of the MP paradigm. Many of the specific

intrusion detection techniques a CuPIDS will use differ from their StUPIDS counterparts only in the real-time, simultaneous monitoring nature of their use. Of particular interest to us are those efforts that separate runtime error checking from runtime execution, those modeling the state of the production process externally, and those making use of coprocessors or virtual machine architectures in performing monitoring tasks. This section presents only a sampling of the relevant literature given in [7]

2.3.1 Debugging

An example from the separate runtime error checking body of research is that done by Patil and Fischer [8] on detecting runtime errors in array and pointer accesses. They point out that including runtime error checking may slow applications by as much as a factor of 10, which is an enormous price to pay given that most runs of a well-tested program are error-free. Therefore once debugging and testing is complete, runtime error checks are disabled before the code is placed into production use. While this makes sense from a performance perspective, it is dangerous because errors that may have been caught by those runtime checks go undetected, potentially causing severe damage. The authors responded by creating guard programs that model the execution of the production program, but only at the pointer and array access level. The guards include all runtime checks on pointer and array bounds and were capable of detecting many runtime errors that evaded the software testers during development. These guards were run as batch processes using trace information stored by the production process. The paper also discussed having the guard run on a separate processor or as a normal process, interleaving execution with the production process. The runtime penalty perceived by the user was typically less than 10%. We use the idea of exporting runtime checks to a shadow process; however, our work differs from theirs in that we focus on real-time monitoring of the actual memory locations in use by the production process as well as a much larger set of monitoring capabilities.

2.3.2 External Modeling

Research into performing intrusion detection via external modeling of application behavior such as the work done by Haizhi Xu et al. in using context-sensitive monitoring of process control flows to detect errors is a good example of external modeling [9]. They define a series of "waypoints" as points along a normal flow of execution that a process must take. They focused their efforts on the system call interface and demonstrated good results in detecting attempts to access system resources by a subverted process. CuPIDS makes use of a similar idea to their waypoints in its checkpoints, those points in both the interactive and passive systems where CuPIDS is notified of events in which

it is interested; however, CuPIDS checkpoints are much finer-grained and are generated within the production process as well as its interaction with the external environment. As an example, CuPIDS uses function call entry and exit information to perform rough granularity program counter tracking and validation as well as model a program stack for use in detecting illegitimate control flows within a process code segment.

Related work by Feng et al. [10] describes novel work in extracting return addresses from the call stack and using abstract execution path checking between pairs of points to detect attacks. Finally, Gopalakrishna et al. [11] present good results in performing online flow- and context-sensitive modeling of program behavior. Gopalakrishna's Inlined Automaton Model (IAM) addresses inefficiencies in earlier context-sensitive models [12,13] by using inlined function call nodes to dramatically reduce the non-determinism in their model while applying compaction techniques to reduce the model's memory usage. Using an event stream generated by library call interpositioning, IAM is shown to be efficient and scalable even in a StUPIDS architecture. The techniques used by IAM fit naturally into the CuPIDS architecture. The model simulation can be run as a CSP, getting its inputs from the CuPIDS event streams.

2.3.3 Virtualization and Co-processors

ID has been performed using both machine virtualization and the use of dedicated co-processors [1,2,14–17]. An example of the latter category includes the work done by Zhang, et al. in describing how a crypto co-processor is used to perform some host-based intrusion detection tasks[15]. In their research they examine the possible effectiveness of using hardware designed for securely booting the system to run an intrusion detection system. The benefits from doing so include protecting the IDS processor from the production processor, and offloading IDS work from the main processor onto one dedicated for that task. Strengths of this approach include high attack resistance for code running in the co-processor system. Drawbacks of the approach include the lack of ready visibility into the actions of the main processor and operating system.

These strengths and drawbacks also exist in the use of virtual machine architectures. Garfinkel and Rosenblum discuss a novel approach to protecting IDS components [17]. They pull the IDS out of the host and place it in the virtual machine monitor (VMM) with the primary goal of enhancing attack resistance. This approach has the benefit of largely isolating the IDS from code running in the virtual host. The VMM approach has much in common with the reference monitor work discussed by Anderson [18] and Lipton [19] in that it provides a means by which the IDS can mediate access between software running in the virtual host and the hardware. It can also interpose at the architecture interface, which yields a better view into the system operation

by providing visibility into both software and hardware events. A traditional software-only IDS does not have this advantage. Of course, the IDS running in the VMM has visibility only into hardware-level state. This means that the IDS can see physical pages and hardware registers, but must be able to determine what meaning the host O/S is placing on those hardware items. By running as part of the host O/S, CuPIDS maintains complete visibility into the software state of the entire system, but currently lacks the protection afforded VMs and secure co-processor architectures. Future work on CuPIDS will use hardware protection mechanisms such as those in the Intel IA32 [20] processor line to provide protection of security specific components as well as critical operating system components.

3 CuPIDS Architecture

The CuPIDS architecture was initially presented in [21], and is summarized here.

3.1 High Level Design

CuPIDS operates using the facilities and capabilities afforded by a general purpose symmetrical multi-processing (SMP) computer architecture. Common operating systems such as Windows, Linux, and FreeBSD running on SMP architectures use the CPUs symmetrically, attempting to allocate tasks equally across the CPUs based upon system loading [22]. CuPIDS differs from these architectures in that at any point in time one or more of the CPUs in a system are used exclusively for security related tasks. This asymmetrical use of processors in a SMP architecture is a significant departure from normal computing models, and represents a shift in priority from performance, where as many CPU cycles as possible are used for production tasks, to security where a significant portion of the CPU cycles available in a system are dedicated solely to protective work. One possible CuPIDS software architecture is depicted in Figure 1. The dark components represent production tasks and services and run on one CPU while the light components represent the CuPIDS monitors and run on a separate CPU. The regions of overlap depict CuPIDS ability to monitor the resource usage of production components.

The operating system as well as user processes are divided into components that are intended to run on separate CPUs. The intent behind this separation is twofold: performance, where we seek to minimize the runtime penalty imposed by the security system, and protection, where we are concerned with the completeness of detection. By ensuring the processes responsible for detecting

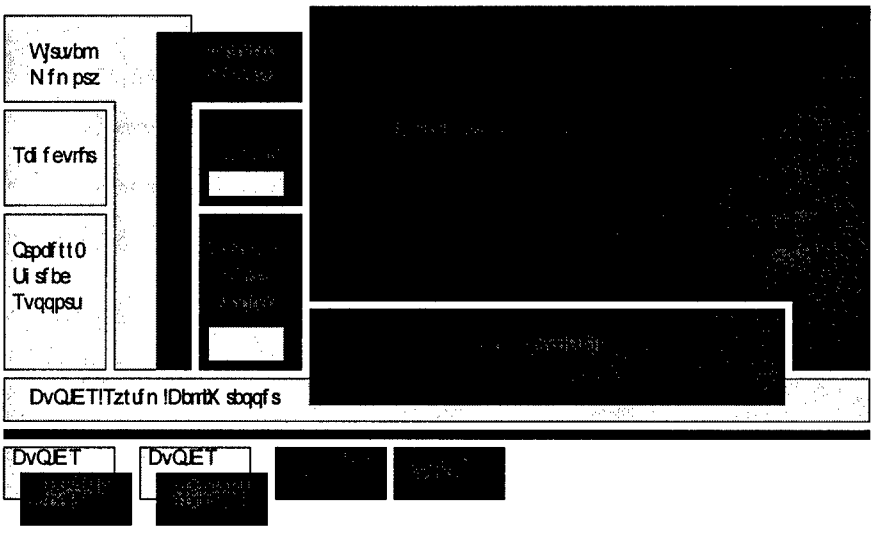

Fig. 1. Basic software architecture

bad events are actively monitoring the system during periods in which bad events can occur—the CuPIDS architecture requires that when a CPP is executing its associated CSP is also on a CPU—we provide a real-time detection capability (using Kuperman's notation as defined in Section 2.1). The system protection derives in part from the ability to detect bad events as they occur but before the results of these events can cause a system compromise.

A program intended to operate in CuPIDS is divided into two components, a CuPIDS monitored production process (CPP) and a shadowing CuPIDS process (CSP) as depicted in Figure 2.

As the figure shows, CuPIDS processes differ from the traditional process paradigm in the asymmetric sharing of memory between the CSP and CPP. The CPP is a normal process and contains the code and data structures that are used to accomplish the tasks for which the program is designed. It may also contain code and data structures with which information about the state of the running process is communicated to the security component. In addition to the normal process code and data structures, the CSP's virtual memory is modified to contain portions of the CPP's virtual memory space (depicted in the figure as Shadow Memory). This allows the CSP to directly monitor the activities of the production component as it executes.

Our initial work assumes the CPP developer is aware of CuPIDS and the CPP communicates its state to the CSP by sending a stream of messages about events of interest to CuPIDS. Later work will investigate what types of real-time monitoring are possible for uninstrumented applications.

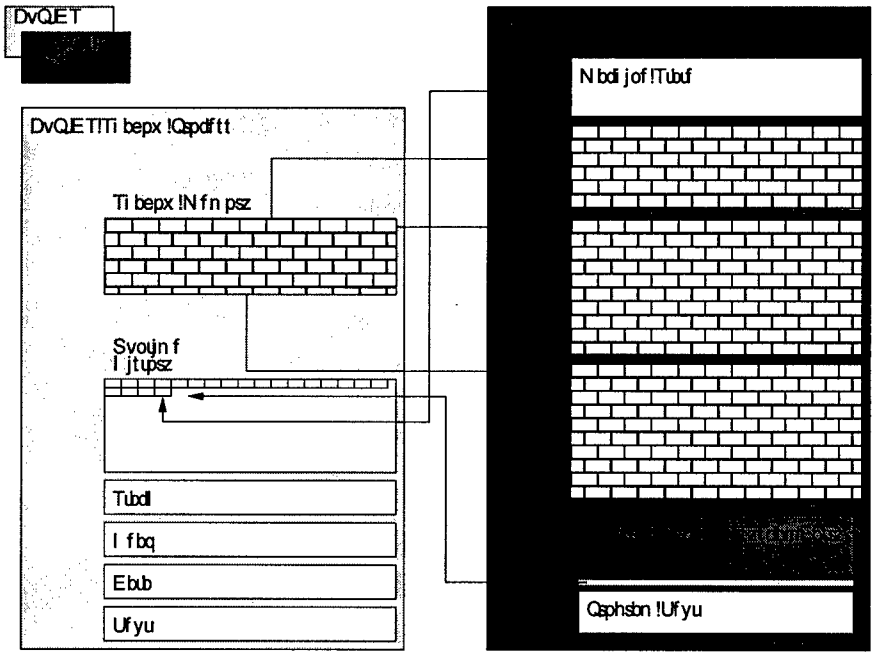

Fig. 2. CSP and CPP details

3.2 Protective Activities

The CuPIDS architecture currently supports three types of protective activities: Application startup/shutdown validation, state monitoring, including invariant testing, and execution monitoring.

Application startup/shutdown: Startup tasks include verifying the authenticity of both the CSP and CPP as well as any supporting configuration files. The CSP is loaded and started executing. It then loads the CPP into memory, establishes any needed hooks into the CPP's VM space, initializes the various event communication systems, and finally starts the CPP running. Shutdown tasks include verifying that the CPP shutdown path followed a legitimate code path. Additionally, any runtime history data is saved to disk.

State monitoring Assertion verification: By creating appropriate hooks into the kernel CSP is able to monitor nearly all aspects of the CPP's operating environment and state, including its entire VM space and any related kernel data structures and excluding only the internal processor state while the CPP is on a CPU. One use of this capability is invariant testing. Invariant testing is a two stage process involving pre-compilation work and runtime invariant checking. The pre-compilation task involves determining which variables need monitoring, defining invariants for those variables and exporting that information in a form that can be used by the CSP. The

compiler is also used to automatically instrument the CPP by adding event generation hooks into each function prologue and epilogue. Invariants are currently snippets of code which could be directly included in the CPP's code (similar to the run-time debugging tests discussed earlier). They are compiled into the CSP's code, and when one is used, it is given appropriate pointers into the CPP's virtual memory space and executed. Currently these are manually written; however, work is underway to allow a programmer to indicate, via pragmas, to the compiler that a particular variable needs protection and the compiler will automatically generate the invariant testing code in the CSP.

Runtime execution monitoring Runtime monitoring includes a number of activities and capabilities that give the CSP visibility into the operation of the CPP. An example includes generating events so the CSP is made aware of the creation, accesses to, and deletion of a protected variable's lifespan. Other events export an execution trace to CuPIDS via function call monitoring, and interactions between the CPP and external environmental entities such as calls to runtime libraries and the operating system. Call monitoring consists of the CPP sending a stream of function/library/system call entry and exit events to the CSP. The CSP then uses a model based upon how the CPP is supposed to operate to verify if that stream is legitimate.

In addition to the direct monitoring of the CPP performed by the CSP, CuPIDS has a number of background capabilities that augment the CSP's capabilities. These include the ability to intercept and direct low-level system activities such as interrupts and signals, controlling the system scheduler to enforce the segregation of the CuPIDS and system CPUs and ensuring that whenever a CSP is chosen to run, its associated CSP is also placed on the CuPIDS' CPU. Additionally, CuPIDS provides a very streamlined, interrupt-based, communication interface for moving event records from the CPP to the CSP running on a different CPU.

3.3 Self-healing/Self-protection

There are a number of well-known-to-be-dangerous library and syscalls [23]. Among the most common exploits publicly available are buffer overflows which use unsafe string handling library functions to overflow vulnerable buffers. Using a combination of stack modeling, library call event monitoring and the virtual memory mapping capability it is possible for CuPIDS to automatically detect and generate detection signatures for certain common classes of vulnerabilities such as stack-based overflows. In many cases buffer overflows use known library function such as strcopy(3). When CuPIDS is notified of a call to strcopy it can create a copy on write (COW) mapping of the page(s) containing the buffer and surrounding memory region. If information about buffer

sizes is available to the CSP, either automatically generated or inserted by the programmer in the form of CuPIDS memory operation events it becomes possible for CuPIDS to not only detect and generate signatures for anomalous events, but also to recover from them automatically. It does so by using the saved copy of stack (or heap) pages to recreate the process' memory state as it was before the overflow, and copying only the correct amount of data into the buffer from the corrupted pages. While in the case of an exploit attempt the data ending up in the buffer may not be what the CPP programmer intended, the overall effect to the program is the same as if a safe string copy function such as strncopy(3) had been used. In addition, error variables or signals may be set to indicate that something unexpected occurred.

4 Implementation

We have implemented a prototype CuPIDS. This section briefly describes the current state of that prototype. Our experimentation uses FreeBSD, currently 5.3-RELEASE [24]. We have added to the operating system API a set of CuPIDS specific system calls that give CuPIDS processes visibility into and control over the execution of a CPP. Examples of the new functionality include the ability to map an arbitrary portion of the PP's address space into the address space of a CSP, a means by which signals destined for and some interrupts caused by the CPP are routed to the monitoring CSP, etc. The operating system kernel has been modified to perform the simultaneous task switching of CPPs and CSPs, a CSP protected loading capability as discussed above in section 3.2, and hooks into various kernel data structures have been added to allow the CSP better visibility into CPP operation and for runtime history data gathering.

Our experimentation to date has focused on protecting specific applications [2]. We perform interactive monitoring based upon automatically generated instrumentation from the compiler as well as CPP programmer defined invariants for key variables. CuPIDS has the capability to examine program binaries and extract explicit white-lists about which system resources are used by the CPP, and then save this information in a form usable by the CSP. As the CPP runs it sends messages to the CSP notifying it about operational activities such as protected variable lifetime events (creation, accesses and deletion) as well as control flow events (currently all function call entry and exits, to include library and syscall invocations) are passed to the CSP as well. The CSP receives these messages and uses them to ensure the CPP is operating correctly. In the case of variables the CSP performs pre- and post-condition

[2] The techniques involved are largely applicable to operating system protection as well

invariant checking, and in the case of flow control, it verifies that all function calls are to and from legitimate locations within the CPP text segment. It also maintains a model of the CPP call stack and verifies all function returns are to the correct locations, etc.

We have used this prototype to verify basic CuPIDS functionality. The system is able to correctly load and execute CPP and CSP components, the CSP is able to detect invariant and security policy violations as well as illegitimate control flow changes. Upon detecting a fault or attack, the CSP is able to halt the PP, raise an alarm, save the state of the CPP's memory and execution trace history, and in some cases repair the damage from the attack or error, allowing the CPP to continue execution without interruption. Time-related testing results are discussed below.

5 Results

The experiments described here demonstrate that it is possible for one process to efficiently perform realtime runtime error checking on variables in another process as well as perform simple flow control validation. To demonstrate the validity of our research hypothesis we demonstrate that CuPIDS can provide guaranteed detection of certain attacks before a context switching event occurs. This claim cannot be matched by a StUPIDS, even if equipped with a comparable detector set.

In our experimentation we used a combination of widely-used, open source applications and servers as well as applications created specifically to test certain aspects of CuPIDS' functionality. The commonly used applications were WU-FTP version 2.6.2 and gnats version 3.113.12. These programs were chosen because they represent software typical of that used in our target environment, their source code is available so that we could examine and instrument them, and because they contain exploitable vulnerabilities as demonstrated by publicly available zero-day exploits.

WU-FTP's ftpd daemon was used to perform performance measurements of CuPIDS as well as to test CuPIDS' invariant violation detection and self-healing capabilities. The ftpd daemon was ideal for this purpose because it is fairly large (about 20000 lines of code), its behavior is representative of many server-type applications in that it runs for long periods and forks off child processes to handle requests, and finally because it has a number of buffer overflow vulnerabilities[3].

[3] CVE entries CVE-1999-0878, CAN-2003-0466, and CVE-1999-0368 [25]

We used gnats-3.113.12 because of the existence of a locally exploitable vulnerability[4]. gnats was used to test CuPIDS' ability to detect invariant violations.

We used the CuPIDS prototype to perform a number of experiments, allowing us to demonstrate the validity of our research hypothesis. A detailed discussion of those results is available in [7], and is summarized here.

5.1 Test Platform

The experiments described below were run on a MP platform with dual Xeon 2.2GHz processors, 1G RAM, 1 120GB ATA100 drive. Hyperthreading (HTT) was enabled so the operating system had available 4 CPUs. We recognize that the performance of HTT processors does not match that of separate CPUs[20]; however, the architecture is useful to us for other reasons. While the results discussed here do not make use of HTT specific features we use do make use of the fact that they share architectural components in research discussed in [7]. For experiments involving CuPIDS, the CSP is the only user of CPU1, the instrumented ftpd uses all of one CPU's cycles, and the ftp client uses all of another CPU's cycles, and the system, including the test drivers, mostly run on the fourth CPU. The test drivers ensure that all file I/O is done on local drives so that network overhead does not become a factor. During the non-instrumented experiments CPU1 is held idle to provide ftpd the same operating environment as it had in the instrumented runs. ftpd was run as root in standalone mode (command line `ftpd -s` which causes it to stay in the foreground and fork processes as needed).

5.2 Runtime Efficiency Tests using WU-FTP

The initial experiments connect to the ftp daemon, log in, and perform 300 ftp file transfers and one ls for a total of 301 transfers. The file transfer workload is 1881832400 bytes and the overall workload per experiment is 1881904317 bytes. Three sets of five experimental runs were made, one using the CuPIDS interrupt-based IPC, one using SysV IPC, and one baseline test was run against an non-instrumented version of WU-FTP. The results are summarized in Table 1.

The initial tests are intended to measure the overhead involved in getting CuPIDS events out of the CPP and into the CSP, therefore we constructed a worst-case event load based on program flow control monitoring. In the instrumented tests, all function calls generate entry and exit events. This

[4] CVE CAN-2004-0623[25]

Event Communications Method	Real Time (seconds)	User Time (seconds)	System Time (seconds)	Throughput (MB/seconds)
Interrupt-based Avg	130.02	0.50	1.51	14.97
Interrupt-based Std Dev	0.52	0.07	0.06	0.05
SysV IPC-based Avg	241.38	0.44	1.81	13.77
SysV IPC-based Std Dev	2.19	0.02	0.06	0.11
Non-Instrumented Avg	118.50	0.38	1.86	16.16
Non-Instrumented Std Dev	0.10	0.03	0.10	0.02

Table 1
WU-FTP Runtime Performance Measurements

includes internal functions, libc and intra-libc calls as well as system calls. Each event includes caller address and callee address information. These events are validated against a white list of calls statically extracted from the ftpd binary. The initial white list contained all the legitimate non-function-pointer-based function and shared library calls as well as a list of all function pointer uses. An initial experimental run identical to the timing runs was made to train the CSP on the actual function pointer usage. The CSP received each function/library/system call event, verified it against the white list, and used it to model the CSP's program stack. The timing related tests did not include embedded invariant tests.

Each experimental run took between two and four minutes and generated approximately 1.4 million events corresponding to WU-FTP's activities. As shown in Table 1 the overhead of generating and using those events was less than ten percent for the CuPIDS IPC as opposed to approximately 100 percent for the SysV-based IPC. Note that this overhead should be balanced against the removal of an inline IDS doing the same tasks. Even a standalone IDS with a similar detector set would be competing for CPU cycles with the CPP, likely degrading application performance.

5.3 Control Flow Change Results

A number of experiments were run to validate CuPIDS' ability to detect illegitimate control flows in the CPP. A summary of those experiments is presented here and described in more detail in [7].

- Illegitimate Syscall Detection: Both gnats and WU-FTP were used in these tests. In both applications a buffer was overflowed in such a way that byte-code contained in the overflow string was executed. The injected code made a number of system calls from the stack. CuPIDS was able to detect all of

the illegitimate system calls.

- Illegitimate Internal Function Call Detection: Both gnats and WU-FTP were used in these tests. In both applications CuPIDS was able to detect an internal function call that had been removed from the white list (simulating the activity of injected code that makes calls to functionality embedded in the vulnerable application). CuPIDS was also able to detect calls into functions that bypassed the prologue event generator. It did so by detecting illegitimate program stack activity in the stack model.
- Illegitimate Library Call Detection: Both gnats and WU-FTP were used in these tests. In both applications CuPIDS was able to catch a call to a library function which was removed from the white list.
- Spoofing/Masquerading Detection: CuPIDS detected attempts to make library or system calls from locations other than those specified in the white lists. This prevents attackers from performing masquerading attacks such as those described in [26]. The CuPIDS IPC mechanism guards against spoofed event generation by including in each event the return address for the generating function as taken from the stack. As the address is placed on the stack by the processor and reading it occurs in kernel space there is no way for a user program to spoof this information.
- Direct Variable Protection: WU-FTP was used for these experiments, which involved performing invariant testing on simple variables (int, char, simple structs) and a string buffer. As discussed earlier, CuPIDS was able to detect illegitimate changes to both classes of variables. In the case of a stack-based buffer overflow it was able to detect the overflow, save the overflowing data, repair the corruption to memory following the buffer, terminate the string in the buffer appropriately (by writing a zero into the end of the buffer), allow the CPP to continue running, and write the overflow string and information about the overflow out to disk. In these experiments the detection took place as the overflow occurred, so CuPIDS was able to halt the CPP before it could return into the corrupted instruction pointer on the stack. Therefore the attack was stopped before any control flow change took place–a capability unique to a parallel monitoring architecture like CuPIDS. Even had the buffer overflow not been directly detected, CuPIDS would have detected the control flow change to the stack and may have been able to make the same repair.

5.4 Desired Supportive Capabilities

While the results presented above show promise, we believe that a paradigm shift towards multi-processor security may lead to changes in the basic platform upon which architectures like CuPIDS are built. Some areas we anticipate exploring include:

Compiler support The compiler can automatically generate events for variable lifecycle operations. As an example, as buffers are allocated and used appropriate events can be generated and dispatched. Another alternative is to allow the programmer to direct the compiler to do this work using a mechanism such as pragma, or assertions.

Hardware support Better support for moving blocks of information between specific CPUs will be useful. As an example, the shared registers on the Xeon HTT processors provide a convenient scratchpad for small amounts of information. Additionally, better debugging capabilities can be designed. A capability similar to the debug registers but on shared memory, and possibly on larger data areas would be useful. The ability to set a memory write breakpoint on a CSP CPU and have it detect writes to that memory location by other CPUs would reduce the number of messages needed to keep track of CPP activity. It may be more practical to do this type of operation on multicore processors.

Operating System Support More efficient means of IPC designed specifically around an asymmetrical MP design such as CuPIDS are possible. CuPIDS' extensions to the FreeBSD API are a start in this direction, and the extended inter-processor-interrupt (IPI) message passing system from the DragonFly BSD variant [27] would probably be useful.

5.5 Self-protection

Mandatory access control (MAC) models such as Biba's integrity-based model [28], and Bell and LaPadula's multi-level security [29] models might be used to provide a first-line defense against user application compromise. While MAC protection systems are not novel, the CuPIDS architecture uses hardware protection mechanisms in commodity CPUs to define and protect the MAC mechanism and CuPIDS themselves against direct attacks that attempt to bypass its controls.

6 Conclusion

For many information systems, high assurance, in terms of keeping an application running in spite of faults or attacks, is more important than raw performance. This is particularly true for an organization's mission critical applications and servers. We believe and demonstrate that dedicating one or more processors in a MP system specifically to security tasks can increase system robustness in the face of faults and attacks. We further believe offloading the security work from the production parts of the system will allow the use of security techniques which may be too computationally expensive when

performed inline.

Examples of such techniques include the runtime debugging checks and assertions employed during the software development process. Checks such as these are commonly placed in vulnerable or critical code during the debugging and testing phases of the software lifecycle but are removed from shipping code because the runtime performance degradation they impose [8]. A great deal of specifically focused information about how an application is intended to behave is available to system architects and developers; however, we do not believe this wealth of information is commonly used in runtime security monitoring of production systems. The CuPIDS architecture is specifically designed to make use of such information in a reasonably efficient manner.

We have proposed a paradigm shift in computer security, one that challenges conventional wisdom by trading performance for security. Our approach is based upon running dedicated monitoring functions in parallel with the code they monitor on a MP system. We believe the CuPIDS architecture to be more effective than StUPIDS architectures in terms of real-time detection of bad events as well as offering some novel detection techniques based upon the low-level and parallel nature of the monitoring. By dedicating computational resources explicitly to security tasks we are trading performance for security; however, by offloading some security tasks from the production process into the security process and running them in parallel we are decreasing the workload of the system production components. We have constructed a prototype of this architecture and used it to verify CuPIDS basic functionality.

The CuPIDS architecture is novel in that we explicitly divide the system into production and security components, embed explicit knowledge of how the production components are intended to operate into specialized security monitors and ensure the appropriate security component is running on a processor whenever a particular production component is running on a different processor. The architecture allows fine-grained visibility into the operation of a protected process. We intend the CuPIDS architecture to be detection model agnostic—capable of supporting many different IDS.

The detection capability of CuPIDS is currently all specification or white-list-based. Therefore it has a zero false positive error rate; thus the alarms output from CuPIDS are suitable for use by automated response systems. In fact, much of CuPIDS strength derives from its automated response capabilities. Its tightly focused, parallel monitoring capability allows for rapid detection of and response to illegitimate behavior. The combination of real-time detection (discussed in Section 2.1) allowed by parallel processing and an ability to automatically repair some damage afforded by CuPIDS' low-level interface into the host operating system let CuPIDS not only stop the attacks, but help maintain operations of critical components of systems.

References

[1] J. D. Tygar, B. Yee, Dyad: A system for using physically secure coprocessors, in: IP Workshop Proceedings, 1994.
URL citeseer.nj.nec.com/tygar91dyad.html

[2] W. A. Arbaugh, D. J. Farber, J. M. Smith, A secure and reliable bootstrap architecture, in: In Proceedings 1997 IEEE Symposium on Security and Privacy, pages 65–71, May 1997., 1997.
URL citeseer.nj.nec.com/arbaugh97secure.html

[3] M. Crosbie, et. al., Idiot users guide, Tech. Rep. COAST TR 96-04, Department of Computer Sciences, cSD-TR-96-050 (1996).
URL
https://www.cerias.purdue.edu/techreports-ssl/public/96-04.ps

[4] B. A. Kuperman, A categorization of computer security monitoring systems and the impact on the design of audit sources, Ph.D. thesis, Purdue University, West Lafayette, IN, cERIAS TR 2004-26 (08 2004).

[5] P. Manadhata, J. M. Wing, Measuring a system's attack surface, Tech. Rep. CMU-CS-04-102, CERIAS, Purdue University, Pittsburgh, PA (January 2004).

[6] S. Axelsson, Intrusion detection systems: A survey and taxonomy, Tech. Rep. 99-15, Chalmers Univ. (Mar. 2000).
URL citeseer.nj.nec.com/axelsson00intrusion.html

[7] P. D. Williams, CuPIDS: Co-processor based intrusion detection system, Ph.D. thesis, Purdue University, West Lafayette, IN, not yet complete (08 2005).

[8] H. Patil, C. Fischer, Low-cost, concurrent checking of pointer and array accesses in C programs, Softw. Pract. Exper. 27 (1) (1997) 87–110.

[9] H. Xu, W. Du, S. J. Chapin, Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths, 2004, in Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection.

[10] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, W. Gong, Anomaly detection using call stack information, in: SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy, IEEE Computer Society, 2003, p. 62.

[11] R. Gopalakrishna, E. H. Spafford, J. Vitek, Efficient intrusion detection using automaton inlining, in: Proceedings of the 2005 IEEE Symposium on Security and Privacy, IEEE Computer Society, 2005.

[12] D. Wagner, D. Dean, Intrusion detection via static analysis, in: SP '01: Proceedings of the IEEE Symposium on Security and Privacy, IEEE Computer Society, 2001, p. 156.

[13] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, B. P. Miller, Formalizing sensitivity in static analysis for intrusion detection., in: IEEE Symposium on Security and Privacy, 2004.

[14] O. S. Saydjari, LOCK: An Historical Perspective, in: Proceedings of the 18th Annual Computer Security Applications Conference, 2000, ACSAC, www.acsac.org, 2000, pp. Online, www.acsac.org.

[15] X. Zhang, L. van Doom, T. Jaeger, R. Perez, R. Sailer, Secure coprocessor-based intrusion detection, in: ACM European SIGOPS 2002, 2002.
URL `www.research.ibm.com/vali/sigops2002_monitor.ps`

[16] J. Molina, W. A. Arbaugh, Using independent auditors as intrusion detection systems, in: Proceedings of the Fourth International Conference on Information and Communications Security (S. Qing, F. Bao, and J. Zhou, eds.), Vol. 2513 of LNCS, 2002, pp. 291–302.

[17] T. Garfinkel, M. Rosenblum, A virtual machine introspection based architecture for intrusion detection, in: Proc. Network and Distributed Systems Security Symposium, 2003.
URL `citeseer.nj.nec.com/garfinkel03virtual.html`

[18] J. P. Anderson, Computer security technology planning study, Tech. Rep. ESD-TR-73-51, Vol. II, HQ Electronic Systems Division (AFSC), Hanscom Field, Bedford, MA, 01730 (1972).

[19] R. Lipton, S. Rajagopalan, D. Serpanos, Spy: A method to secure clients for network services, in: Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops, 2002.

[20] I. Tm-I, Ia-32 Intel architecture software developers manual volume 1: Basic architecture.
URL `developer.intel.com/design/pentium4/manuals/245472.htm`

[21] P. D. Williams, E. H. Spafford, CuPIDS enhances StUPIDS: Exploring a co-processing paradigm shift in information system security, in: To Appear: Proceedings of the 6th IEEE Information Assurance Workshop, IEEE, 2005.

[22] A. Silberschatz, P. B. Galvin, G. Gagne, Operating System Concepts, John Wiley & Sons, Inc., 2001.

[23] G. Hoglund, G. McGraw, Exploiting Software: How to Break Code, Pearson Higher Education, 2004.

[24] TrustedBSD, TrustedBSD, www.freebsd.org.

[25] Mitre, Common vulnerabilities and exposures.
URL `www.cve.mitre.org`

[26] T. Ptacek, T. Newsham, Insertion, evasion, and denial of service: Eluding network intrusion detection, Tech. rep., Secure Networks, Inc. (1998).

[27] DragonFlyBSD, DragonFlyBSD, www.dragonflybsd.org.

[28] K. Biba, Integrity considerations for secure computer systems, Tech. Rep. TR-3153, Mitre, Bedford, MA (Apr. 1977).

[29] D. E. Bell, L. J. LaPadula, Secure computer systems: Mathematical foundations and model, Tech. Rep. M74-244, The MITRE Corp., Bedford MA (May 1973).

THE VIEWS EXPRESSED IN THIS ARTICLE ARE THOSE OF THE AUTHOR AND DO NOT REFLECT THE OFFICIAL POLICY OR POSITION OF THE UNITED STATES AIR FORCE, DEPARTMENT OF DEFENSE, OR THE U.S. GOVERNMENT.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	12.Jul.05	MAJOR REPORT

4. TITLE AND SUBTITLE
CUPIDS: AN EXPLORATION OF HIGHLY FOCUSED, CO-PROCESSOR-BASED INFORMATION SYSTEM PROTECTION

5. FUNDING NUMBERS

6. AUTHOR(S)
CAPT WILLIAMS PAUL D

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
PURDUE UNIVERSITY

8. PERFORMING ORGANIZATION REPORT NUMBER
CI04-1129

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
THE DEPARTMENT OF THE AIR FORCE
AFIT/CIA, BLDG 125
2950 P STREET
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unlimited distribution
In Accordance With AFI 35-205/AFIT Sup 1

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS

15. NUMBER OF PAGES
20

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT